Janus: An Active Middleware for Accessing Sensor Networks

No Author Given

No Institute Given

Abstract. One of the research challenges in sensor networking is how to access the resources of the sensor network from a remote location i.e., collecting information and reacting to events from the sensor network. Although access mechanisms already exist, they are typically application-specific and thus inflexible. This reduces the potential life-time of the sensor network by limiting how we can change it after deployment. We may wish to upgrade the sensor network after deployment due to changing application requirements and other unforeseen circumstances. Sensor network deployments in harsh or remote environments make upgrading very difficult. We present an architecture called Janus which comprises of an extensible middleware for interfacing with a sensor network and a code deployment platform which can be run inside of a sensor network. We model the sensor network as a collection of functions which provides us with a natural way to extend its functionality by adding more functions. We have implemented the middleware and tested it against a sample real-world sensor network and have implemented the code deployment platform in the Contiki network simulator.

1 Introduction

Sensor networks are an attractive solution to many remote monitoring applications such as environmental, habitat and intrusion monitoring scenarios. In order for a sensor network to be a viable solution to monitoring a harsh or isolated environment where it is hard to physically access the sensors, there must be some technique to access the resources of the sensor network remotely. How a sensor network is accessed depends a great deal on how the sensor network is represented i.e., how it appears to the outside world. A typical approach is to provide a database abstraction of the sensor network [1]. However, we chose to model the sensor network as a collection of functions. We call this approach function-orientated. This provides us with a natural way to extend the capabilities of the sensor network as a new piece of functionality can be represented as one more function in the sensor network.

The motivation for extending a sensor network’s functionality through a remote mechanism is that once a sensor network is deployed, changing it manually through physical access is often very difficult. This is because there may be many sensors, we may not actually know where these sensors are located and physical access to the location where the sensors are may be infeasible. We may wish to upgrade the sensor network to patch bugs in the deployed system, the existing software or even add new functionality to the system to take into account changing application requirements. One sensor network deployment which is used for habitat monitoring is the “Great Duck
Island” system [2] which is located on an island off the coast of Maine, USA. The bird colonies on this island are extremely sensitive to disturbance and researchers are typically only on-site during the summer months. This means that upgrading a sensor node or network without a remote mechanism is extremely difficult. How to achieve remote upgrading is therefore the focus of this paper.

We present an architecture called Janus which comprises of an extensible middleware for providing access to the sensor network resources from an external network (previously presented in a workshop paper from the authors) and a code deployment platform which runs inside the sensor network. We acknowledge that security is a major issue with such a code deployment approach. We imagine that existing security tools for active networks could be applied to our architecture as well. For example: Proof Carrying Code (PCC) [3], programming language support [4] or a more holistic approach [5]. The contribution of this paper is the demonstration of how a sensor network can be represented as a collection of functions and how this approach allows us to deploy code in the sensor network and access it externally in a uniform way. Uniformity comes from Janus using the same protocols inside the sensor network as it does outside the sensor network. Since we wish to use the same protocol both inside and outside of the sensor network, it is a requirement that the protocol is lightweight, this makes it difficult to reuse existing signaling protocols such as SIP [6]. The goal of our approach is to provide as much freedom as possible to the sensor network designers as to how they wish to implement their solutions. We do not present new protocols or algorithms for data diffusion or aggregation inside of the sensor network, but rather present an architecture where it is possible to deploy new functionality both inside and outside the sensor network in a straightforward manner. Whilst there already exist work on Active Sensor Networks [7] and Mobile Agent Middleware [8] which provide remote code deployment to a sensor network, our approach differs in three main ways: 1. we use a function-orientated approach to model the sensor network i.e., we represent the sensor network as a collection of functions, 2. we do not use a virtual machine such as Maté [9] and 3. we provide a uniform query/response mechanism to interact with the middleware outside of the sensor network as well as the code deployed in the sensor network. We have implemented the middleware portion of Janus on an example sensor network and the code deployment portion of Janus in the Contiki [10] network simulator.

The rest of the paper is organized as follows: we first present the architecture of Janus and our implementation work. We then discuss some related work and describe our ideas for future work. We close with our conclusions.

2 Architecture

The design of the Janus architecture is presented in figure [1]. Janus comprises of three entities:

1. The eXtensible Resolution Protocol (XRP) which the engine and the agent use to communicate with each other.

---

1 Janus is the two-faced Roman god of doorways
2. an *XRP engine* (henceforth referred to as the engine), which runs either on the sink node in the sensor network or in the sensor network itself.

3. an *XRP agent* (henceforth referred to as the agent) that communicates with the engine.

![Diagram of Janus architecture](image)

**Fig. 1.** Janus architecture. Janus itself is comprised of the XRP Agent, the XRP Engine and the XRP signaling protocol. The example HTTP, SMS gateway & SQL servers are collectively known as Front Ends servers.

The agent and the engine exchange XRP messages with each other in order to:

1. Dynamically discover available sensor network resources. This enables the agent to ascertain which resources are available to it via the engine. This is useful in situations where the agent does not have explicit knowledge of the exact nature of the sensor network resources. This is also used to discover new functions which have been deployed dynamically into the sensor network.

2. Send queries from the agent to the gateway concerning the state of sensor network. Examples of these involve querying the current temperature, light levels and motion sensing.

3. Send information from the gateway to the agent about the state of the sensor network. An agent can be notified by the sensor network when a specific event occurs. Examples of these include when motion is detected or when there is a significant change in temperature.

4. Push functions into the sensor network which can then be used as described in 1. and 2.
The gateway uses XRP to export the sensor network resources as functions which the agent can access. The agent can then invoke these functions by sending XRP messages to the gateway. The functions are named by selectors, which are opaque identifiers and uniquely map to functions at the gateway. When a new function is added to the sensor network, its selector is added to the nodes in the sensor network and/or to the engine so that it can be accessed in the same way as existing functions can.

2.1 Function Placement

One of the key design decisions of the Janus architecture is to attempt to have as much choice as possible about where to place functionality. We note here that, for us, a sensor network system is not just the sensor network itself but also the mechanisms that allow the sensor network to be manipulated from an external location. In our architecture, functionality can be deployed in the following places:

**Front Ends**: The front ends are the part of the Janus architecture which interface directly with the user. As show in figure 1, examples of which are HTTP servers or SMS gateways. Although they are not exactly part of the Janus architecture they still need to be able to interface with it. A new front end can be added to a deployed system by adding the ability to process the application-specific front end protocol to the agent.

**Agent**: The Agent is typically located outside of the sensor network and is therefore suitable to be upgraded as it does not have the typical sensor network deployment restrictions of power consumption, processing, memory or connectivity. We imagine the Agent to usually reside on a commodity PC or equivalent. Since the Agent is located outside of the sensor network, we believe that upgrading this single entity manually to be a straightforward task.

**Engine**: The Engine is located at the edge of the sensor network. In certain deployments we can imagine the Engine residing on a lightweight PC or PDA where access to power is not an issue. Other deployments, such as those in animal habitats, may not have this luxury. In such situations we imagine the engine to run on the sensor network node which is functioning as a sink. Adding functionality to the Engine dynamically from a remote location is a crucial part of Janus. This is described in section 3.3. The same approach applies for every node in the sensor network including the sink.

2.2 XRP – eXtensible Resolution Protocol

The eXtensible Resolution Protocol (XRP) defines a presentation format for the internal Janus messages. It permits to specify a variable and extensible number of parameter fields. The parameter format resembles the encoding of RSVP objects [11]. An XRP message is a sequence of XRP commands. Each command is introduced by a 32-bit command header:
followed by zero or more command parameters. Command parameters have the format:

```
<table>
<thead>
<tr>
<th>0</th>
<th>length (in bytes)</th>
<th>class</th>
<th>class-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>... contents ...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The XRP command parameters are of variable length and their semantics is given by a `class` field, which names the parameter and a `class-type` which defines the parameter’s data type. XRP parameters can appear anywhere inside the parameter block, thus are not bound to a specific position. See previous work by the authors for a more detailed description of the XRP packet format and its possible uses. XRP is used by Janus to communicate between the agent and the engine as well as inside of the sensor network itself. Janus uses XRP as a basis upon which to construct an RPC-style interface to the sensor network.

### 2.3 Agent

The role of the agent is to exchange XRP messages with the engine to query the resources of the sensor network. The agent exchanges XRP messages with the engine in order to dynamically discover available sensor network resources, to send queries to the gateway concerning the state of the sensor network and to send or receive information from the gateway about the events in the sensor network. Figure 2 shows a client that has connected to an agent in order to send queries to the sensor network. The agent acts as the translation point between the client and the engine to provide access to the sensor network resources.

Janus uses dynamic discovery of sensor network resources by letting the agent negotiate available services with the engine on the gateway. This is particularly useful in situations where the agent does not have explicit knowledge of the exact nature of a sensor network. It enables the agent to find out which resources are available to it via the engine. One of the main advantages of the Janus architecture is that the same agent can be used to access different sensor networks without being forced to implement this explicit application-specific knowledge, as shown in Figure 1 as long as the same API is implemented at the engine. After the API has been negotiated between the agent and the gateway the agent can issue queries to the sensor network through the gateway in a way similar to the RPC client-server model. This is done by sending a request XRP messages over the network targeting one of the selectors exported with the API. Examples of these requests involve querying the current temperature, humidity levels and motion sensing. Through the interface offered by the gateway it is also possible for an agent to register an interest for a particular type of event within the sensor network. The agent will then be notified when this specific event occurs. Examples of these events include when motion is detected anywhere inside the network or when there is a significant change in temperature in a given region. The agent can be invoked by separate front ends to provide user interaction and present data to existing clients such as web browsers.
Fig. 2. The client sends queries to the agent (1) which will translate these into XRP commands to an engine (2). When the reply has been acquired from the sensor network the engine will reply (3) with an XRP message that is translated and sent to the client (4).

2.4 Engine

The engine is in charge of the agent interaction on the gateway. This interaction takes the form of XRP messages sent back and forth between the agent and the engine. When an agent has connected to the gateway it begins to issue XRP commands. These commands are interpreted by the engine. The agent can discover the available sensor network resources in the gateway by sending a request for the API to the function module on the gateway. The engine processes this request and composes an XRP message with the API represented as special XRP parameters. This XRP message is then transmitted back to the agent. This messages contains a list of the selectors bound to the individual functions of the function module on the gateway.

Functions are invoked on the gateway when the engine processes an XRP message targeting one of these selectors in a way similar to the RPC-style invocation. When a local function is triggered it will communicate directly with the sensor network to retrieve the answer to the posed question, build a reply message containing this answer and transmit it back to the agent. In some sense the engine can be viewed as a proxy interface to the sensor network. This enables us to provide a uniform interface to the sensor network by enabling us to abstract away the internals of the sensor network if required.

The engine is also capable of installing newly defined functions dynamically which are sent via XRP messages. Subsequently these functions can then be referenced in the same way as existing functions.

2.5 Gateway

The gateway sits between the sensor network and the access network and contains an engine as seen in Figure 2. Its role is to handle incoming connections from agents in the access network and to listen to incoming events from the sensor network. When an agent connects to the gateway it can begin to issue XRP commands, which are interpreted by the engine at the gateway.

For every sensor network accessed with the Janus architecture there would have to be at least one gateway. This is to provide the sensor network with an ingress / egress
point in which it can be accessed. If the need arises to create a gateway for a previously unsupported sensor network only the access and function modules of the gateway risk to be coded. In other words it would be necessary to implement an access method to be able to communicate with the network and functions which can be used to query the sensor hardware on the actual sensor nodes. Due to the dynamic nature of the Janus architecture adding a new sensor network type like this would not warrant any changes outside the gateway. If we decide to upgrade our nodes with a new type of sensors, we would only have to change the functionality module and could leave the access module intact.

The functionality module contains all the functions which can be used to query the sensor network. These functions can be exported using XRP to allow an agent to access and query the sensor network. Simplified the API is represented as a list of the available functions and the corresponding selectors used to invoke them. The list is implemented using the special API-style XRP messages and is described in more detail in Section 3.4.

3 Implementation

We have implemented a prototype as a proof-of-concept which comprises of a subsection of the architecture running against a sample sensor network. We stress here that this is just an example of how a sensor network could be implemented, there are many different ways that this could actually be performed. Our prototype was written in C++ under Linux and we connected it to our sensor network using a USB connector attached to a sink node. We have successfully run tests with the agent and the gateway + engine on separate machines and with a sensor network of up to 16 sensor nodes.

3.1 Agent

In our implementation a client connects to the agent using Berkeley TCP sockets. To be able to offer the client the ability to actively query the sensor network the agent must first request and parse the API of the gateway in a way explained in Section 3.2. The

![Fig. 3. An XRP message comes in from an agent in the Janus network (1) and are received by the gateway which hands it over to the engine for processing. The engine invokes a packet processing function which queries the sensor network (2). The reply comes back from the sensor network (3) and is passed on to the agent (4).](image)
agent can also register an interest to receive event notification from the sensor network by sending an XRP SUBSCRIBE message to the gateway. An agent can either subscribe to events of a given verbosity (e.g., silent, critical, verbose, very verbose) or to specific events (e.g., motion in a given area). The engine will then add the agent to its list of subscribers and inform it when an event takes place.

In our current implementation the location (i.e., IP address) of the gateway is statically defined, but in the Janus architecture there would ideally be some kind of service discovery to locate gateways in your vicinity. This could be done using broadcasted XRP resolution requests in a way similar to the one described in previous work by the authors.

### 3.2 Engine

The agent can discover the available sensor network resources on this gateway by sending an XRP message containing a QUERY for the API to the gateway. This process is shown in Figure 5. When our implemented engine receives this request it will respond with a pre-built XRP message containing a DATA REPLY with the API. After this initial handshaking the agent will be able to invoke functions at the engine by sending QUERY style messages for the selectors listed in the API. When this happens the QUERY is demultiplexed in the engine and a local function invoked with the data payload as a function argument. In our implementation the function argument is either the address of a node (position 1-16 in a grid) or the alias of an area (e.g., beach, field, forest and selected) inside the network. The local function communicates with the sensor network and returns a result. An XRP DATA REPLY message is then built and sent back to the agent.

### 3.3 Code Deployment Platform

Our code deployment platform (shown in figure 4), which is designed to be run on a sensor network node, uses the engine described above. Currently we have an implementation of the code deployment platform running in Contiki network simulator and we are planning to port it to our example sensor network platform. The code deployment platform interfaces with the Contiki OS at the link layer i.e., before the network layer processing takes place. Janus packets are diverted from the typical packet processing path as show on the left-hand side of Figure 4 and into the Janus function table. Currently our implementation provides two main services:

**Multi-hop ad-hoc routing and forwarding**: This functionality is provided through the µLUNAR system that we have ported to Contiki. This allows us to communicate point-to-point over multi-hop inside of the sensor network if we need to communicate with a specific node rather than a function for debugging and testing purposes. Future application scenarios may even require such a communication approach.

**Code deployment**: Code deployment using µLUNAR to forward code-carrying packets through the network. We broadcast XRP messages into the network which carry a piece of code to be installed and define which area we wish them to be valid in.
Contiki currently divides the sensor network into a geographic area addressable by $x$ and $y$ co-ordinates. This is useful for reprogramming a certain part of the network. The code is then installed at the appropriate locations in the network and a selector is returned to the source (usually the sink, although this is not necessary) which allows this newly installed functionality to be invoked.

Our current unoptimized version of the code deployment platform is 8KB in size and we expect that this size can be further reduced.

### 3.4 Signaling between Agent and Engine

All XRP messages in our implementation are transmitted via UDP between the agent and the engine residing on the gateway. All the functions to interact with the sensor network are implemented at the gateway. We provide both low-level functions to query individual nodes and high-level functions to allow for more advanced queries, like the mean value inside a given area.

The gateway listens to events that originate inside the sensor network and waits on incoming connections to reach its engine. Once an agent connects, it negotiates with the gateway to retrieve an API of functions to reach the sensor network. After the API has been exported to the agent, it can issue requests by invoking functions at the gateway.

The signaling is achieved using a combination of XRP DREQ (Data Request) and DREP (Data Reply) messages as shown in figure 5. The figure shows how XRP packets are sent between the agent and the engine at the gateway. A selector to handle the reply is always installed prior to sending a request and we will refer to this selector as the
**Fig. 5.** Signaling: The left column in the table represents function addresses and the right table column represents function names. (1,2) Exporting the API of the sensor network (3,4) Issuing an RPC by addressing a selector bound to a function together with the packet payload as an argument.

reply selector. Each DREQ contains the reply selector along with the address of the sending node. This information is used by the gateway to target its reply.

When an agent wishes to retrieve the API from a gateway, it installs a reply selector \( r \) and issues:

1. a DREQ for the API to the gateway. The engine running at the gateway then processes the request
2. The engine subsequently responds with a DREP to the \( r \) selector on the agent. This reply contains the API represented as XRP parameters inside the packet. In figure 5 we show the `query_max()` function along with its selector. This function takes an area as the argument and returns the node with the maximum sensor value. After the API has been exported, the agent can invoke functions at the gateway by addressing selectors on the gateway with a QUERY style message. The agent installs a reply selector \( s \) to issue
3. a DREQ with the function argument as the payload, in this case the requested area inside the network. The engine running at the gateway processes this request by invoking the corresponding local function. The gateway issues individual requests for the sensors within the given area to find the node with the highest sensor value. The engine proceeds by building a DREP containing the returned value as an XRP parameter
4. The engine then sends this to the reply selector at the agent.
3.5 Sample Sensor network

The sample sensor network used in the prototype implementation was built on sensor boards developed at Freie Universität in Berlin as shown in figure 6. Technical details regarding the sensor boards can be found at the Scatterweb ESB home page [12]. The sensors can detect motion, light, temperature, vibration and sound. The sensors nodes run Contiki [10], a small operating system for tiny devices which allowed us to implement the desired functionality in C. We have implemented a network where nodes store sensed sound data, aggregate it and deliver it to the edge of the network in response to individual queries. The sensors also notice if they are moved and this special event is then automatically reported. Connectivity is supported by attaching a USB serial cable to the sink node.

![Fig. 6. An ESB3 ScatterWeb sensor node](image)

4 Related Work

There is relevant related work in the areas of introducing active networks and mobile agents to sensor networks and access techniques for sensor networks.

4.1 Active Network and Mobile Agents in Sensor Networks

There has been recent work on Active Sensor Networks [7] and Mobile Agent Middleware [8]. Active Sensor Networks present an extension to the Maté [9] virtual machine system to provide application-specific virtual machines (ASVMs) which are virtual machines customized for specific tasks. The focus of this work is to provide an appropriate programming abstraction for programming sensor networks. Janus does not use a virtual machine approach, instead relying on the programming environment provided by the Contiki OS. Additionally, the ASVM approach does not provide a middleware which provides an abstraction of the sensor network. We are interested in providing a complete architecture for accessing sensor networks. Mobile Agent Middleware provides a mobile agent platform called Agilla which enables agents to proactively crawl
the sensor network and spawn to appropriate locations to execute. The Agilla approach uses a tuple space on the sensor network node which the mobile agents can use to communicate and store data. We do not wish to provide such a detailed abstraction with our work, but rather provide the support to deploy such systems as Agilla which provide a certain set of tools to the sensor network designer.

4.2 Sensor Network Access Techniques

TinyDB [1] is perhaps the most well-known example of a sensor network database abstraction. TinyDB makes the sensor network appear as a database that can be queried for data. Data queries are made using an SQL-like syntax, and data is transported through the sensor network back to a gateway node. Data aggregation is used to reduce the amount of communication in the network. Whilst TinyDB provides some hooks to extend its SQL-like syntax, the extensibility of the system is not its fundamental goal. We envisage TinyDB being one of many systems that could be deployed through the Janus approach.

5 Future Work

We wish to explore the following topics in future work: Firstly, performance profiling and power consumption are extremely important to the viability of Janus. We need to quantify the impact of our architecture on the operational efficiency of the sensor network. This will enable us to understand the trade-offs inherent in deploying such a system. Secondly, security is a major issue for active networking approaches and for sensor networks. Without comprehensive answers to these questions, we expect sensor networks to remain a purely research issue, limited to a few experimental deployments. Existing research in active network security [3, 4, 5] point out a set of directions that we should pursue. There also exist proposals for security in sensor networks which should also be carefully examined [13].

6 Conclusions

We have presented Janus, an architecture for accessing sensor network resources which comprises of an extensible middleware located outside of the sensor network and a code deployment platform which resides inside of the sensor network. Our architecture provides a uniform query/response API to access the sensor network resources. Uniformity is achieved by using the same protocols both inside and outside of the sensor network. Our goal is to have a system for providing access to sensor network resources which is both extensible and lightweight. The system needs to be extensible so that it can be upgraded and modified during deployment without physical access to the sensor network itself since many typical sensor network deployments are in harsh or remote conditions. How lightweight the system is, is also a concern as sensor networks typically are extremely resource-constrained and do not have access to external power sources. Janus presents a view of the sensor network as a set of functions which can be invoked
on-demand or can trigger responses to events. We have implemented a prototype of our architecture and successfully run tests which demonstrate typical sensor network scenarios.

References